

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

Отчет
по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»»
Тема: Иерархические списки

Студент гр. 8304

Завражин Д.Г.

Преподаватель

Фирсов М. А.

Санкт-Петербург

2019

Цель работы.

Ознакомиться с основными понятиями и приёмами программной реализации иерархических списков, освоить навыки разработки и написания процедур их обработки на языке C++ на примере поставленного задания.

Задание.

Вариант 33

Пусть *арифметическое* выражение представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в *постфиксной* форме (<аргументы> <операция>). Аргументов может быть 1, 2 и более. Например (в префиксной форме): (+ a (* b (- c))).

В задании даётся следующий вариант требуемого действия с выражением: *проверка синтаксической корректности и деления на 0*.

В индивидуальном задании указывается: тип выражения (возможно дополнительно - состав операций), вариант действия и форма записи. Указания, соответствующие данному варианту, подставлены в текст задания и выделены курсивом.

Класс *TrivariateHierarchicalList*

С целью программной реализации структуры данных иерархического списка на основе указателей была создана структура данных *TrivariateHierarchicalList*, представляющая собой иерархический список, каждый элемент которого хранит либо указатель на подсписок, либо одит из задаваемых шаблоном типов данных.

Интерфейс класса *TrivariateHierarchicalList* состоит из:

- Конструктора, инициализирующего первый элемент списка;

- Деструктора, используемого для освобождения памяти;
- Метода `represent`, возвращающего строковое представление иерархического списка;
- Метода `begin`, возвращающего итератор начала списка;
- Метода `end`, возвращающего итератор конца списка.

Они имеют следующие сигнатуры:

- `TrivariateHierarchicalList()`
- `TrivariateHierarchicalList()`
- `std::string represent()`
- `Iterator begin()`
- `Iterator end()`

Класс *TrivariateHierarchicalList* имеет два недоступных извне вложенных класса: *Node* и *Iterator*.

Реализация класса *TrivariateHierarchicalList* и вложенных в него классов, находящаяся в заголовочном файле *trivariatehierarchicallist.h*, приведена вместе со всем исходным кодом программы в приложении А.

Класс *Node*

Класс *Node* реализует один элемент иерархического списка и написан на основе типа данных `std::variant` из стандартной библиотеки языка C++17. Его интерфейс состоит из:

- Конструктора;

- Деструктора, используемого для освобождения памяти;
- Геттера `next` и сеттера `setNext`, обеспечивающих доступ к указателю, указывающему на следующий элемент списка;
- Геттера `content` и сеттера `setContent`, обеспечивающих доступ к содержимому элемента списка;
- Метода `represent`, возвращающего строковое представление иерархического списка с данным элементом в качестве головного.

Сигнатуры методов класса *Node*:

- `Node()`
- `void setNext(Node* const next)`
- `const std::variant<T, U, V, Node*>& content()`
- `void setContent(const T& content)`
- `void setContent(const U& content)`
- `void setContent(const V& content)`
- `void setContent(Node* const content)`
- `std::string represent()`

Класс *Iterator*

Класс *Iterator* реализует нерекурсивную итерацию по атомарным элементам иерархического списка. Его интерфейс состоит из:

- Конструктора, запоминающего указатель, указывающий на хранимый в нём элемент и переходящий к первому атомарному после него;

- Унарных операторов инкремента;
- Бинарных операторов равенства и неравенства;
- Метода `getPreviousNodeCount`, возвращающего количество элементов на том же уровне до него.

Заметим, что освобождение памяти по указателю, хранящемуся у данного итератора, не требуется, так как эта память будет освобождена при вызове деструктора класса *TrivariateHierarchicalList*.

У него также имеется недоступный извне метод `normalizePosition`, реализующий переход к следующему атомарному элементу.

Методы класса *Iterator* имеют следующие сигнатуры:

- `Iterator(Node *node=nullptr)`
- `Iterator& operator++()`
- `Iterator operator++(int)`
- `Node& operator* ()`
- `Node* operator-> ()`
- `bool operator==(const Iterator& that)`
- `bool operator!=(const Iterator& that)`
- `size_t getPreviousNodeCount()`
- `void normalizePosition()`

Класс *Expression*

С целью программной реализации структуры данных для представления строки данного в условии задания вида на основе иерархического списка была создана структура данных *Expression*, представляющая собой подкласс иерархического списка с добавлением необходимых для работы с выражениями методов.

Интерфейс класса *Expression*, вдобавок к интерфейсу класса *TrivariateHierarchicalList*, состоит из:

- Конструктора, принимающего строку;
- Метода `isCorrect`, проверяющего корректность выражения;
- Метода `getErrors`, возвращающего список ошибок в выражении.

Вдобавок к этому, он имеет следующие недоступные извне методы:

- Рекурсивного метода `parse`, преобразующего переданную конструктором строку в иерархический список;
- Итеративного метода `checkNodes`, проводящего проверку вычислимости выражения и деления на 0 путём попытки вычисления выражения на стеке без подстановки неизвестных переменных.

Методы класса *Expression* имеют следующие сигнатуры:

- `Expression(const std::string& expression)`
- `bool isCorrect()`
- `std::string getErrors()`
- `std::string::const_iterator parse(ExpressionNode *node, std::string::const_iterator current, std::string::const_iterator end)`

- `void checkNodes(std::stack<std::variant<T,bool>>& executionStack)`

Ошибки, обнаруженные в некотором выражении, хранятся в классе в двух векторах строк, `parsingErrors` и `executionErrors`, соответственно хранящие ошибки, обнаруженные при выполнении методов `parse` и `checkNodes`.

Реализация класса *Expression*, находящаяся в заголовочном файле *expression.h*, приведена вместе со всем исходным кодом программы в приложении А.

Метод *parse*

Метод *parse* класса *Expression* выполняет рекурсивное преобразование строки типа `std::string`, чьи итераторы переданы ему, в иерархический список, представленный данным классом; при этом метод вызывается только для каждого уровня списка.

Метод *checkNodes*

Метод *checkNodes* класса *Expression* выполняет итеративную проверку выполнимости хранящегося в классе выражения. При этом проводятся проверки на:

- Соответствие количества операндов операции;
- Постфиксность выражения;
- Деление на 0.

Используемый алгоритм для избежания рекурсии прибегает к описанному ранее итератору *TrivariateHierarchicalList::Iterator* и непосредственным действиям со стеком. В процессе вычислений промежуточные значения хранятся как `std::variant<T,bool>`, где `T` — тип числовых данных, а в типе данных `bool` хранятся недоступные для вычисления данные, которые могут быть:

- Значениями переменных;
- Результатами вычисления содержащих ошибки операций.

Вспомогательные функции и типы данных

В процессе выполнения работы были также созданы:

- Класс перечисления `lab2::OperationType`, для хранения кодов операции;
- Функция `lab2::to_string`, эквивалентная функции `std::to_string`, перегруженной для типов данных `std::string` и класса перечисления `lab2::OperationType`;
- Функция `lab2::stoi`, приводящая строковый тип данных `std::string` к заданному целочисленному формату.

Их реализация, находящаяся в заголовочном файле `utils.h`, приведена вместе со всем исходным кодом программы в приложении А.

Функция *main*

Функция *main* выполняет задачу получения от пользователя строки, содержащей анализируемое выражение. Это может происходить двумя способами:

- Посредством передачи в качестве единственного аргумента командной строки;
- Посредством ввода по прямому запросу программы.

После получения пути программа передаёт его функции конструктору класса `Expression`, выполняющему вызов его метода `parse`, обеспечивающего преобразование строки в иерархический список.

Сигнатура функции `main`: `int main(int argc, char* argv[])`.

Тестирование программы

Тесты, содержащиеся в файле *tests.txt* и важные с точки зрения оценки работ программы фрагменты её вывода, приведены в таблице 1.

Таблица 1 – Тесты, применённые при тестирование программы.

№	Тест	Обнаруженные ошибки
1		The provided expression is empty
2	()	The provided expression contains empty parentheses
2	(3 a)	There is 1 unused operand left.
2	(5 6 +)	The string contains the following character(s) after the last closing bracket: ")"
2	(5 6 +) 5	The string contains the following character(s) after the last closing bracket: " 5"
2	((5 6 +) 5)	There is 1 unused operands left.
2	(5 0 /)	Term 3: Division by 0 encountered.
2	(5 (5 5 -) /)	Term 5: Division by 0 encountered.
2	((6 (5 9 *) /) (4 (6 5/) -) +)	The given expression is correct.
2	((6 (5 9 *) /) (4 (6 0/) -) +)	Term 5: Division by 0 encountered.
2	((x1 (x2 x3 *) /)(x4 (x2 (5.5 x3*) /) -) +)	The given expression is correct.

На всех приведённых выше входных данных программа выдаёт ожидаемый результат; отсюда можно сделать вывод, что данная программа корректно

работает во всех охватываемых составленными тестами случаях.

Вывод

В результате выполнения лабораторной работы была реализована программа, отвечающая всем поставленным условиям и проходящая рассмотренное выше составленное в процессе выполнения работы тестирование. Помимо этого, были на практическом примере отточены навыки проектирования, написания и тестирования иерархических списков и алгоритмов работы с ними, владения языком C++.

Приложение А

Исходный код программы

Программа, использованная при выполнении лабораторной работы и представленная ниже, написана на языке программирования C++.

utils.h

```
1  #ifndef LAB2_UTILS_H_
2  #define LAB2_UTILS_H_
3
4  #include <string>
5
6  namespace lab2
7  {
8      // enum class OperationType is used to encode an arithmetic
       operation
9      enum class OperationType {ADDITION, SUBTRACTION, MULTIPLICATION
       , DIVISION};
10
11     // lab2::to_string is similar to std::to_string, but is
       overloaded for two
12     // additional types
13     std::string to_string(const std::string& string)
14     {
15         return string;
16     }
17
18     std::string to_string(lab2::OperationType operationType)
19     {
```

```

20     if(operationType == lab2::OperationType::ADDITION)
21         return "+";
22     if(operationType == lab2::OperationType::SUBTRACTION)
23         return "-";
24     if(operationType == lab2::OperationType::MULTIPLICATION)
25         return "*";
26     if(operationType == lab2::OperationType::DIVISION)
27         return "/";
28     return "?";
29 }
30
31 std::string to_string(auto value)
32 {
33     return std::to_string(value);
34 }
35
36 // lab2::stoT is a generalization of std::stoi, std::stol, ...
37 // for a numeric type T
38 template<class T>
39 T stoT(std::string);
40 template<>
41 short stoT<short>(std::string str){return std::stoi(str);}
42 template<>
43 int stoT<int>(std::string str){return std::stoi(str);}
44 template<>
45 long stoT<long>(std::string str){return std::stol(str);}
46 template<>
47 long long stoT<long long>(std::string str){return std::stoll(
    str);}
48 template<>
49 unsigned short stoT<unsigned short>(std::string str){return std
    ::stoul(str);}
50 template<>

```

```

51     unsigned int stoT<unsigned int>(std::string str){return std::
        stoul(str);}
52     template<>
53     unsigned long stoT<unsigned long>(std::string str){return std::
        stoul(str);}
54     template<>
55     unsigned long long stoT<unsigned long long>(std::string str){
        return std::stoull(str);}
56     template<>
57     float stoT<float>(std::string str){return std::stof(str);}
58     template<>
59     double stoT<double>(std::string str){return std::stod(str);}
60     template<>
61     long double stoT<long double>(std::string str){return std::
        stold(str);}
62 }
63
64 #endif // LAB2_UTILS_H_

```

trivariatehierarchicallist.h

```

1  #ifndef LAB2_TRIVARIATEHIERARCHICALLIST_H_
2  #define LAB2_TRIVARIATEHIERARCHICALLIST_H_
3
4  #include <variant>
5  #include <stack>
6
7  #include "utils.h"
8
9  namespace lab2
10 {

```

```

11      // TrivariateHierarchicalList class was designed to store
12      either a value of
13      // one out of three distinct types or a pointer to a sublist
14      template<class T, class U, class V>
15      class TrivariateHierarchicalList
16      {
17      protected:
18          class Node;
19      private:
20          class Iterator;
21
22      public:
23          explicit TrivariateHierarchicalList()
24          {
25              this->head_ = new Node();
26          }
27
28          ~TrivariateHierarchicalList()
29          {
30              delete this->head_;
31          }
32
33          std::string represent()
34          {
35              return this->head_->represent();
36          }
37
38          Iterator begin()
39          {
40              return Iterator(this->head_);
41          }
42
43          Iterator end()

```

```

43         {
44             return Iterator(nullptr);
45         }
46
47     protected:
48         Node* head()
49         {
50             return this->head_;
51         }
52
53     private:
54         Node* head_ = nullptr;
55
56     protected:
57         // The nested class Node was designed to be a single node
58         of a
59         // trivariate hierarchical list
60         class Node
61         {
62         public:
63             explicit Node()
64             {
65                 this->content_ = nullptr;
66             }
67
68             ~Node()
69             {
70                 if(std::holds_alternative<Node*>(content_))
71                     delete std::get<Node*>(content_);
72                 delete this->next_;
73             }
74
75             Node* next()

```

```

75         {
76             return this->next_;
77         }
78
79     void setNext(Node* const next)
80     {
81         this->next_ = next;
82     }
83
84     const std::variant<T, U, V, Node*>& content()
85     {
86         return this->content_;
87     }
88
89     void setContent(const T& content)
90     {
91         this->content_ = content;
92     }
93
94     void setContent(const U& content)
95     {
96         this->content_ = content;
97     }
98
99     void setContent(const V& content)
100    {
101        this->content_ = content;
102    }
103
104    void setContent(Node* const content)
105    {
106        this->content_ = content;
107    }

```



```

108
109         std::string represent()
110     {
111         std::string representation = "(";
112         auto current = this;
113         while(current != nullptr)
114         {
115             if(std::holds_alternative<Node*>(current->
116                 content_) &&
117                 std::get<Node*>(current->content()) !=
118                     nullptr)
119                 representation +=
120                     std::get<Node*>(current->content_->
121                         represent());
122             else if(std::holds_alternative<T>(current->
123                 content_))
124                 representation +=
125                     lab2::to_string(std::get<T>(current->
126                         content_));
127             else if(std::holds_alternative<U>(current->
128                 content_))
129                 representation +=
130                     lab2::to_string(std::get<U>(current->
131                         content_));
132             else if(std::holds_alternative<V>(current->
133                 content_))
134                 representation +=
135                     lab2::to_string(std::get<V>(current->
136                         content_));
137             if(current->next_ != nullptr)
138                 representation += ' ';
139             current = current->next_;
140         }
141     }

```

```

132         return representation + '>';
133     }
134
135     private:
136         std::variant<T, U, V, Node*> content_;
137         Node *next_ = nullptr;
138     };
139
140     private:
141         // The nested class Iterator was designed to facilitate
142         iteration
143         // through a hierarchical list using explicit stack
144         manipulations
145         class Iterator
146         {
147         public:
148             using difference_type = std::ptrdiff_t;
149             using value_type = Node;
150             using pointer = Node*;
151             using reference = Node&;
152             using iterator_category = std::forward_iterator_tag;
153
154             explicit Iterator(Node *node)
155             {
156                 this->current = node;
157                 this->previousNodeCountStack.push(0);
158                 this->normalizePosition();
159             }
160
161             Iterator& operator++()
162             {
163                 if(this->current == nullptr)
164                     return *this;

```

```

163         this->current = this->current->next();
164         previousNodeCountStack.top() += 1;
165         this->normalizePosition();
166         return *this;
167     }
168
169     Iterator operator++(int)
170     {
171         auto old = *this;
172         ++(*this);
173         return old;
174     }
175
176     reference operator *()
177     {
178         return *(this->current);
179     }
180
181     pointer operator ->()
182     {
183         return &**this;
184     }
185
186     bool operator ==(const Iterator& that)
187     {
188         return this->current == that.current &&
189             this->nodeStack == that.nodeStack;
190     }
191
192     bool operator !=(const Iterator& that)
193     {
194         return this->current != that.current ||
195             this->nodeStack != that.nodeStack;

```

```

196         }
197
198     size_t getPreviousNodeCount()
199     {
200         return this->previousNodeCountStack.top();
201     }
202
203     private:
204         std::stack<Node*> nodeStack;
205         std::stack<size_t> previousNodeCountStack;
206         Node* current = nullptr;
207
208         // ensures that this->current holds a pointer to an
209         atomic node
210     void normalizePosition()
211     {
212         while((this->current == nullptr && this->nodeStack.
213             size() > 0) ||
214             (this->current != nullptr &&
215             std::holds_alternative<Node*>(this->current->
216                 content()) &&
217             std::get<Node*>(this->current->content()) !=
218                 nullptr))
219         {
220             while(this->current == nullptr && this->
221                 nodeStack.size() > 0)
222             {
223                 this->current = this->nodeStack.top()->next
224                     ();
225                 this->nodeStack.pop();
226                 this->previousNodeCountStack.pop();
227             }

```

```

222         while(std::holds_alternative<Node*>(this->
                current->content()) &&
223             std::get<Node*>(this->current->content())
                != nullptr)
224         {
225             this->nodeStack.push(this->current);
226             previousNodeCountStack.top() += 1;
227             this->previousNodeCountStack.push(0);
228             this->current = std::get<Node*>(current->
                content());
229         }
230     }
231 }
232 };
233 };
234 }
235 #endif // LAB2_TRIVARIATEHIERARCHICALLIST_H_

```

expression.h

```

1  #ifndef LAB2_EXPRESSION_H_
2  #define LAB2_EXPRESSION_H_
3
4  #include <iostream>
5  #include <vector>
6  #include <regex>
7
8  #include "trivariatehierarchicallist.h"
9
10 constexpr bool DEBUG = true;
11
12 namespace lab2

```

```

13 {
14     template<class T>
15     class Expression
16     : public lab2::TrivariateHierarchicalList<T, std::string, lab2
        ::OperationType>
17     {
18         typedef typename \
19             lab2::TrivariateHierarchicalList<T, std::string, lab2::
                OperationType>::Node \
20             ExpressionNode;
21
22     public:
23         Expression(const std::string &expression)
24         : lab2::TrivariateHierarchicalList<T, std::string, lab2::
            OperationType>()
25         {
26             auto expression_ = regex_replace(expression, std::regex
                ("^\\s*"), "");
27             expression_ = regex_replace(expression_, std::regex("\\
                s*$"), "");
28             expression_ = regex_replace(expression_, std::regex("\\
                s+"), " ");
29             if(DEBUG)
30             {
31                 std::cout << "Acquired string: \"" <<
32                     expression_ << "\"" << std::endl;
33                 std::cout << "Acquired string length: " <<
34                     expression_.length() << std::endl;
35             }
36             if(expression_.length() == 0 || regex_match(expression_
                ,
37
                                                                    std::regex(
                                                                    "\\s*$")

```

```

38         {
39             this->parsingErrors.push_back("The provided
40                 expression is empty");
41         }
42     else
43     {
44         this->parse(this->head(), expression_.begin(),
45             expression_.end());
46     }
47 }
48
49 bool isCorrect()
50 {
51     std::stack<std::variant<T, bool>> executionStack;
52     executionErrors.resize(0);
53     this->checkNodes(executionStack);
54     if(executionStack.size() > 1)
55         this->executionErrors.push_back(std::string("There
56             ") +
57             (executionStack.size() > 2 ? "are " : "is ") +
58             lab2::to_string(executionStack.size() - 1) +
59             " unused operand" + (executionStack.size() > 2
60                 ? "s " :
61                 " ") + "left.");
62     if(this->parsingErrors.size() > 0 || this->
63         executionErrors.size() > 0)
64         return false;
65     return true;
66 }
67
68 std::string getErrors()
69 {

```

```

65         if(this->parsingErrors.size() == 0 &&
66            !(this->executionErrors.size() > 0 || this->
               isCorrect()))
67             return("There are no errors found.\n");
68
69         auto totalLength = 0;
70         for(auto error : this->parsingErrors)
71             totalLength += error.length() + 1;
72         for(auto error : this->executionErrors)
73             totalLength += error.length() + 1;
74
75         std::string result;
76         for(auto error : this->parsingErrors)
77             result += error + "\n";
78         for(auto error : this->executionErrors)
79             result += error + "\n";
80         return "Errors:\n" + result;
81     }
82
83 private:
84     std::vector<std::string> parsingErrors{};
85     std::vector<std::string> executionErrors{};
86
87     std::string::const_iterator parse(ExpressionNode *node,
88                                       std::string::
89                                       const_iterator current
89                                       ,
90                                       std::string::
91                                       const_iterator end)
92     {
93         static size_t depth = 0;
94         depth += 1;
95         if(DEBUG)

```



```

94         {
95             std::cout << std::string(depth - 1, ' ') <<
96                 "|~~~~~" << std::endl;
97             std::cout << std::string(depth - 1, ' ') <<
98                 "| the function \"parse\" was called" << std::
99                 endl;
100             std::cout << std::string(depth - 1, ' ') <<
101                 "| depth: " << depth << std::endl;
102         }
103
104         auto currentNode = node;
105
106         while(current != end && *current == ' ')
107         {
108             current += 1;
109         }
110
111         if(current != end && *current == '(')
112         {
113             if(depth == 1 && DEBUG)
114                 std::cout << std::string(depth - 1, ' ') <<
115                     "| the token \"(\" was acquired: " << std
116                     ::endl;
117             current += 1;
118             while(current != end)
119             {
120                 // skip spaces
121                 while(current != end && *current == ' ')
122                 {
123                     current += 1;
124                 }

```

```

125         {
126             this->parsingErrors.push_back("Expression
127                                     ended unexpectedly");
128             return end;
129         }
130
131         // get "("
132         if(*current == '(')
133         {
134             if(DEBUG)
135             {
136                 std::cout << std::string(depth - 1, ' ')
137                             <<
138                             "| the token \"(\" was acquired: "
139                             << std::endl;
140             }
141             currentNode->setContent(new ExpressionNode
142                                     ());
143             current = parse(
144                 std::get<ExpressionNode*>(currentNode->
145                     content()),
146                 current, end);
147         }
148
149         // get ")"
150         else if(*current == ')')
151         {
152             if(DEBUG)
153             {
154                 std::cout << std::string(depth - 1, ' ')
155                             <<
156                             "| the token \")\" was acquired: "
157                             << std::endl;

```

```

151     }
152     if(std::holds_alternative<ExpressionNode*>(
        currentNode->
153         content())&& std::get<ExpressionNode*>(
            currentNode->
154             content()) == nullptr)
155     {
156         this->parsingErrors.push_back("The
            provided"
157             "expression contains empty
                parentheses");
158     }
159     current += 1;
160     break;
161 }
162
163 // get a variable
164 else if(('A' <= *current && *current <= 'Z') ||
165         *current == '_' ||
166         ('a' <= *current && *current <= 'z'))
167 {
168     std::smatch match;
169     std::regex regex("^([_A-Za-z][_A-Za-z0-9]*)
170         ");
171     std::regex_search (current, end, match,
172         regex);
173     currentNode->setContent(match[1]);
174     if(DEBUG)
175     {
176         std::cout << std::string(depth - 1, ' ')
177             ) <<
178             "| the token \"" <<

```

```

176         std::get<std::string>(currentNode
                                ->content()) <<
177         "\" was acquired." << std::endl;
178     }
179     current += match[1].length();
180 }
181
182 // get a number
183 // fractions of the pattern "[0-9]+" are not
    supported
184 else if('0' <= *current && *current <= '9')
185 {
186     std::smatch match;
187     std::regex regex("^([0-9]+(\\.[0-9]+)?)");
188     std::regex_search (current, end, match,
                        regex);
189     currentNode->setContent(lab2::stoT<T>(match
        [1]));
190     if(DEBUG)
191     {
192         std::cout << std::string(depth - 1, ' ')
        ) <<
193         "| the token \"" <<
194         std::get<T>(currentNode->content()
        ) <<
195         "\" was acquired." << std::endl;
196     }
197     current += match[1].length();
198 }
199
200 // get an operator
201 else if(*current == '+' || *current == '-' || *
    current == '*' ||

```

```

202         *current == '/')
203     {
204         if(*current == '+')
205             currentNode->setContent(lab2::
                OperationType::ADDITION);
206         else if(*current == '-')
207             currentNode->setContent(lab2::
                OperationType::SUBTRACTION);
208         else if(*current == '*')
209             currentNode->setContent(lab2::
                OperationType::MULTIPLICATION);
210         else if(*current == '/')
211             currentNode->setContent(lab2::
                OperationType::DIVISION);
212         current += 1;
213         if(DEBUG)
214         {
215             std::cout << std::string(depth - 1, ' ')
                ) <<
216                 "| the token \"" << lab2::
                    to_string(std::get<\
217                     lab2::OperationType>(currentNode->
                        content())) <<
218                     "\" was acquired." << std::endl;
219         }
220     }
221
222     // ignore other chars
223     else
224     {
225         this->parsingErrors.push_back("Unexpected
                symbol: " +

```

```

226                                                                 lab2::
                                                                    to_string
                                                                    (int(*
                                                                    current))
                                                                    );
227             current += 1;
228         }
229
230         if(current != end && *current != '\n')
231         {
232             currentNode->setNext(new ExpressionNode());
233             currentNode = currentNode->next();
234         }
235     }
236 }
237 else
238 {
239     this->parsingErrors.push_back("The symbol \"(\n\" is
        abscent when necessary");
240 }
241 depth -= 1;
242 if(depth == 0)
243 {
244     // report a presence of a character after the last
        closing bracket
245     if(current != end)
246     {
247         this->parsingErrors.push_back("The string
            contains the \"\n
248             following character(s) after the last closing
            bracket:\n\t\" +
249         std::string(current, end) + "\"");
250     }

```

```

251     }
252     if(DEBUG)
253     {
254         std::cout << std::string(depth, ' ') << "
                |~~~~~" << std::endl;
255         if(depth > 0)
256             std::cout << std::string(depth - 1, ' ') << "|
                depth: " <<
257                 depth << std::endl;
258     }
259     return current;
260 }
261
262 // tries to compute the value of the stored expression in
    order to find
263 // all cases of division by 0
264 // "bool" represents an undecidable variable
265 void checkNodes(std::stack<std::variant<T, bool>>&
    executionStack)
266 {
267     size_t termCount = 1;
268     for(auto current = this->begin(), end = this->end();
269         current != end; termCount += 1, ++current)
270     {
271         // push a number onto the stack
272         if(std::holds_alternative<T>(current->content()))
273         {
274             executionStack.push(std::get<T>(current->
                content()));
275             if(DEBUG)
276             {
277                 std::cout << "~~~~~" << std::endl;

```

```

278         std::cout << "Term " + lab2::to_string(
                termCount) +
279             ", the value is "
280             << std::get<T>(current->content())
281             << std::endl;
282     }
283 }
284 // push an unknown value onto the stack
285 else if(std::holds_alternative<std::string>(current
    ->content()))
286 {
287     executionStack.push(true);
288     if(DEBUG)
289     {
290         std::cout << "~~~~~" << std::endl;
291         std::cout << "Term " + lab2::to_string(
                termCount) +
292             ", the value is undecidable" << std::
                endl;
293     }
294 }
295 // execute an operation
296 else if(std::holds_alternative<lab2::OperationType
    >(current->content()))
297 {
298     auto previousNodeCount = current.
        getPreviousNodeCount();
299     auto operationType = std::get<lab2::
        OperationType>(current->content());
300     if(DEBUG)
301     {
302         std::cout << "~~~~~" << std::endl;

```



```

303         std::cout << "Term " + lab2::to_string(
            termCount) +
304             ", the operation is \"" <<
305             lab2::to_string(operationType) + "\"\"
306             << std::endl;
307     }
308     // if the operation is called as niladic,
        report an error
309     if(previousNodeCount == 0)
310     {
311         this->executionErrors.push_back("Term "
312             + lab2::to_string(termCount) + ": The
            operation \"" +
313             lab2::to_string(operationType) +
314             "\" has too few (0) arguments.");
315     }
316     // if the operation is called as ternary or
        ..., report an error
317     else if(previousNodeCount > 2)
318     {
319         this->executionErrors.push_back("Term "
320             + lab2::to_string(termCount) + ": The
            operation \"" +
321             lab2::to_string (operationType) + "\"
            has too many (" +
322             lab2::to_string(previousNodeCount) + "
            ) arguments.");
323     // remove all its arguments from the stack
324     while(previousNodeCount --> 0)
325     {
326         if(DEBUG)
327         {

```

```

328         std::cout << "Term " + lab2::
           to_string(termCount) +
329             ", the operand " +
330             lab2::to_string(
               previousNodeCount + 1) +
331             " is removed from the stack"
               << std::endl;
332     }
333     executionStack.pop();
334 }
335 // push an unknown value onto the stack as
   its result
336 executionStack.push(true);
337 if(DEBUG)
338 {
339     std::cout << "Term " + lab2::to_string(
       termCount) +
340         ", the result is meaningless" <<
           std::endl;
341 }
342 }
343 // if the operation is called as unary, ...
344 else if(previousNodeCount == 1)
345 {
346     // if it is "-", try to execute it
347     if(operationType == lab2::OperationType::
       SUBTRACTION)
348     {
349         if(std::holds_alternative<T>(
           executionStack.top()))
350         {
351             auto operand = std::get<T>(
               executionStack.top());

```

```

352         if(DEBUG)
353         {
354             std::cout << "Term "
355                 + lab2::to_string(
356                     termCount) +
357                 ", the operand is " <<
358                     operand
359                     << std::endl;
360         }
361         executionStack.pop();
362         executionStack.push(-operand);
363     }
364     else
365     {
366         if(DEBUG)
367         {
368             std::cout << "Term "
369                 + lab2::to_string(
370                     termCount) +
371                 ", the operand is
372                     undecidable"
373                     << std::endl;
374         }
375         executionStack.pop();
376         executionStack.push(true);
377     }
378 }
379 // if it is not "-", report an error
380 else
381 {
382     this->executionErrors.push_back("Term "
383         + lab2::to_string(termCount) +
384         ": The operation \"\" +

```

```

381         lab2::to_string(operationType) +
382         "\" has too few (1) arguments.");
383     executionStack.pop();
384     if(DEBUG)
385     {
386         std::cout << "Term " + lab2::
387             to_string(termCount) +
388             ", the operand " +
389             lab2::to_string(
390                 previousNodeCount + 1) +
391             " is removed from the stack"
392             << std::endl;
393     }
394     executionStack.push(true);
395     if(DEBUG)
396     {
397         std::cout << "Term " + lab2::
398             to_string(termCount) +
399             ", the result is meaningless"
400             << std::endl;
401     }
402 }
403 // if the operation is called as binary, ...
404 else if(previousNodeCount == 2)
405 {
406     // retrieve the operands in reverse order
407     std::variant<T, bool> operand2 =
408         executionStack.top();
409     executionStack.pop();
410     std::variant<T, bool> operand1 =
411         executionStack.top();
412     executionStack.pop();

```

```

407         if(DEBUG)
408         {
409             std::cout << "Term " + lab2::to_string(
                termCount) +
410                 ", the operand 1 is " + (
411                     std::holds_alternative<bool>(
                        operand1) ?
412                         "undecidable" :
413                         lab2::to_string(std::get<T>(
                            operand1)))
414                 << std::endl;
415             std::cout << "Term " + lab2::to_string(
                termCount) +
416                 ", the operand 2 is " + (
417                     std::holds_alternative<bool>(
                        operand2) ?
418                         "undecidable" :
419                         lab2::to_string(std::get<T>(
                            operand2)))
420                 << std::endl;
421         }
422         // check for division by 0
423         if(std::holds_alternative<T>(operand2) &&
424             operationType == lab2::OperationType::
                DIVISION &&
425             std::get<T>(operand2) == 0)
426         {
427             this->executionErrors.push_back("Term "
428                 + lab2::to_string(termCount) +
429                 ": Division by 0 encountered.");
430             executionStack.push(true);
431             if(DEBUG)
432             {

```

```

433         std::cout << "Term " + lab2::
            to_string(termCount) +
434             ", the result is meaningless"
            << std::endl;

435     }
436 }
437 // check for undecidable operands
438 else if(std::holds_alternative<bool>(
    operand1) ||
439         std::holds_alternative<bool>(
            operand2))
440 {
441     executionStack.push(true);
442     if(DEBUG)
443     {
444         std::cout << "Term " + lab2::
            to_string(termCount) +
445             ", the result is undecidable"
            << std::endl;

446     }
447 }
448 // if neither division by 0 nor undecidable
operands were
449 // encountered, execute the operation and
save its result
450 else
451 {
452     T result;
453     if(operationType == lab2::OperationType
        ::ADDITION)
454     {
455         result = std::get<T>(operand1) +
456             std::get<T>(operand2);

```

```

457         }
458         if(operationType == lab2::OperationType
           ::SUBTRACTION)
459         {
460             result = std::get<T>(operand1) -
461                     std::get<T>(operand2);
462         }
463         if(operationType == lab2::OperationType
           ::MULTIPLICATION)
464         {
465             result = std::get<T>(operand1) *
466                     std::get<T>(operand2);
467         }
468         if(operationType == lab2::OperationType
           ::DIVISION)
469         {
470             result = std::get<T>(operand1) /
471                     std::get<T>(operand2);
472         }
473         executionStack.push(result);
474         if(DEBUG)
475         {
476             std::cout << "Term " + lab2::
477                     to_string(termCount) +
478                     ", the result is " +
479                     lab2::to_string(result) << std
480                     ::endl;
481         }
482     }
483     // ignore operands that appear after an
484     // operator
485     if(current->next() != nullptr)

```

```

484         {
485             while(current->next() != nullptr)
486             {
487                 termCount += 1;
488                 ++current;
489                 this->executionErrors.push_back("Term "
490                     +
491                     lab2::to_string(termCount) +
492                     " comes after an operator, and
493                     thus is ignored");
494                 if(DEBUG)
495                 {
496                     std::cout << "~~~~~" << std::endl;
497                     std::cout << "Term " + lab2::
498                         to_string(termCount) +
499                         ", the value is ignored" <<
500                         std::endl;
501                 }
502             }
503         }
504     }
505 };
506 }
507
508 #endif // LAB2_EXPRESSION_H_

```

main.cpp


```

1  #include <iostream>
2  #include <string>
3
4  #include "expression.h"
5
6  using std::cin;
7  using std::cout;
8  using std::endl;
9
10 using lab2::Expression;
11
12 int main(int argc, char* argv[])
13 {
14     if(argc > 2)
15     {
16         cout << "Too many command line arguments were provided." <<
            endl;
17     }
18
19     std::string line;
20     if(argc == 2)
21     {
22         line += std::string(argv[1]);
23     }
24     else
25     {
26         cout << "Enter an expression:" << endl;
27         std::getline(cin, line);
28     }
29
30     Expression<double> expression(line);
31     cout << expression.represent() << endl;
32     if(expression.isCorrect())

```

```

33     {
34         cout << "The given expression \x1b[1mis\x1b[0m correct." <<
            endl;
35     }
36     else
37     {
38         cout << "The given expression is \x1b[1mnot\x1b[0m correct.
            " << endl;
39         cout << expression.getErrors();
40     }
41     cout << endl; // so consecutive calls would be distinct
42 }

```